Kirill Zateishchikov

Scaling a Software Platform Using Micro Frontends

Technology and Communication
2023

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Kirill Zateishchikov
Title	Scaling a Software Platform Using Micro Frontends
Year	2023
Language	English
Pages	38
Name of Supervisor	Mikael Jakas

In the modern era of web development, monolithic frontend applications are often challenged by issues of scale, maintainability, and efficiency. This thesis investigates the innovative approach of using micro frontends, a design pattern which breaks down a frontend monolith into manageable, independent components, thereby improving scalability in the software development process. The primary aim of this thesis is to evaluate the efficiency of micro frontends in improving scalability and efficiency of large-scale web applications.

The methodology employed to explore this paradigm comprises an in-depth study of the core principles of micro frontends, examination of various implementation strategies, and a comprehensive analysis of technical challenges and solutions. A practical case of an industrial automation platform was undertaken to provide practical insights into the application of micro frontends.

The research successfully evaluated the efficiency of micro frontends in improving the scalability of large-scale web applications, resulting in the development of a demo application. This was substantiated by a case study on an industrial automation platform and an in-depth analysis of various implementation strategies and challenges.

Keywords Web programming, software architecture, and JavaScript

CONTENTS

FIGURES AND TABLES

1 INTRODUCTION

1.1 Motivation

The concept of "micro frontends" is a recent innovation in the field of software architecture that aims to tackle the challenges faced by traditional monolithic applications. Similar to microservices, micro frontends promote scalability, availability, and flexibility by breaking down a monolithic application into smaller, independent components.

In the past, monolithic applications were developed as a single entity, but as the codebase grew and new features were added, the development and maintenance of the code became increasingly difficult and time-consuming. This resulted in a rise in software maintenance costs for businesses and made the system increasingly complex to manage (Gilbert, 2021).

Micro frontends aim to solve these challenges by dividing the monolithic application into smaller, self-contained components that are constructed along business capabilities. This provides a clearer view of the functionality and allows for multiple teams to work on different components simultaneously, increasing flexibility and speed of development. The independent deployability of micro frontends also allows for faster and easier updates and maintenance of the system.

However, just like microservices, the use of micro frontends is not always the best solution for every project. Before deciding on this architecture, it is important to consider the benefits and potential drawbacks to ensure that it aligns with the project goals and requirements. Micro frontends can bring advantages such as improved scalability, availability, and flexibility, but also come with trade-offs such as increased complexity and higher development costs.

Micro frontends are a promising solution for addressing the challenges faced by monolithic applications, but it is crucial to carefully weigh the benefits and potential drawbacks before deciding if it is the right architecture for a project. The use

of micro frontends can bring numerous benefits, but it's essential to consider if it aligns with the goals and requirements of the project.

1.2 Scope

The scope of this thesis focuses on the process of transitioning a monolithic Angular application to micro frontends. The transition from a monolithic architecture to micro frontends presents a significant challenge for many organizations. It involves breaking down a large, complex system into smaller, independent components that can be developed and maintained separately. This requires a deep understanding of the existing system, its dependencies, and the business requirements.

1.3 Objective

The objective of this thesis is to provide a comprehensive guide for organizations looking to transition their monolithic Angular applications to micro frontends. This will include an analysis of the current system, identification of suitable micro frontend architecture patterns, and a step-by-step guide for implementing the transition. The thesis will also highlight the benefits and challenges of transitioning to micro frontends, as well as provide recommendations for overcoming these challenges. The end goal is to provide organizations with a clear understanding of the process and the tools required to successfully transition their monolithic Angular applications to micro frontends.

2 MICRO FRONTENDS

2.1 Understanding Micro Frontends

Micro frontends refer to a software development pattern that involves dividing a monolithic frontend application into smaller, self-contained components, thereby breaking down the complexities and dependencies typically associated with large-scale applications. This design approach emphasizes modularity and reusability, enabling developers to create and maintain discrete sections of the user interface with greater ease and efficiency. Each micro frontend is responsible for a specific aspect of the application's user interface, from handling user interactions and displaying content to managing data and providing visual feedback and operates as an independent unit with its own lifecycle, technologies, and deployment strategies (Flower, 2019)

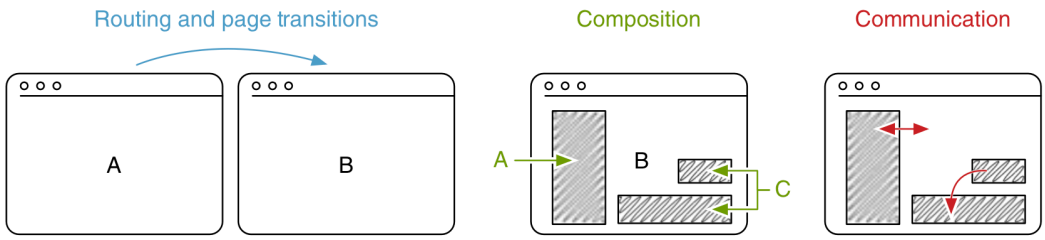

Figure 1. Relationships between Web Components (Geers, 2020).

These components are then combined in the customer's browser to form the final page, offering greater flexibility, scalability, and availability in frontend development by allowing teams to work on different parts of the application simultaneously and with minimal impact on one another. This decoupling of responsibilities also fosters better testing and deployment practices, as each micro frontend can be tested and deployed independently, reducing the risk of failures and bottlenecks associated with monolithic systems. Moreover, the micro frontend architecture promotes innovation and experimentation by giving developers the freedom to choose the best tools and frameworks for each individual component, ultimately leading to a more robust, adaptable, and future-proof frontend ecosystem.

2.2 Micro Frontend Architecture

Micro frontend architecture pattern is similar to the microservices approach, but with a focus on frontend components and their integration with the user interface. Micro frontends provide a new way of developing frontend applications, allowing for faster and more efficient updates, improved collaboration between development teams, and reduced maintenance costs. By breaking down a monolithic frontend application into smaller, manageable components, micro frontends offer a solution to the challenges faced in traditional frontend development (Rappl, 2021).

2.3 Upsides of Micro Frontends

Micro frontends offer several advantages, which make them an attractive choice for organizations looking to enhance their software development process:

1. **Faster development:** Micro frontends streamline the development process by dividing the frontend into smaller, more manageable pieces. This modular approach allows teams to focus on specific features or components and work in parallel, leading to more efficient development and deployment. By breaking down complex tasks into smaller units, teams can achieve a quicker turnaround time for feature releases and bug fixes.
2. **Improved team autonomy:** In a micro frontend architecture, each team works on a distinct portion of the frontend, granting them increased autonomy and control over their work. This independence allows teams to make decisions and implement changes without relying heavily on other teams, fostering creativity and innovation. The self-sufficient nature of these teams enhances overall productivity and reduces bottlenecks in the development process.
3. **Easier scalability:** Micro frontends make it more straightforward to scale applications as they grow. By dividing the frontend into smaller components, each part can be scaled individually, depending on its requirements and usage patterns. This targeted scaling approach helps manage resource

allocation more effectively, resulting in better performance and a more robust application (Geers, 2020).

4. **Incremental upgrades:** Adopting a micro frontend architecture allows organizations to update or replace individual components without affecting the entire application. This granular approach to upgrades facilitates a smoother transition when updating technologies, libraries, or frameworks. As a result, teams can keep their applications up to date without experiencing significant disruptions or downtime.
5. **Better code organization:** Micro frontends promote modular, reusable code and encourage the separation of concerns. By organizing code into smaller, self-contained components, the overall codebase becomes more maintainable, testable, and easier to understand. This improved organization helps minimize technical debt and enhances the long-term stability of the application.
6. **Flexibility in technology choices:** Micro frontends provide the opportunity to utilize different technology stacks for specific components or features. This flexibility allows teams to experiment with new technologies, adopt best practices, and tailor their tools and frameworks to unique requirements of each component. By decoupling technology choices, organizations can innovate more rapidly and adapt to evolving industry trends (Grijzen, 2019).

2.4 Downsides of Micro Frontends

While micro frontends offer various advantages, they also come with potential downsides that organizations must consider before adopting this architectural approach:

1. **Increased complexity:** Micro frontends can introduce additional complexity to the overall system due to the need for managing multiple independ-

ent components, communication between them, and handling deployment and integration. This complexity may require extra effort in planning, coordination, and monitoring.

2. **Performance overhead:** Depending on the implementation, micro frontends can sometimes introduce performance overhead, such as increased latency due to additional network requests or resource loading. Careful attention to performance optimization and resource bundling is necessary to mitigate these issues (Lemon, 2020).
3. **Integration challenges:** Ensuring seamless integration between different micro frontends can be challenging, especially when they are developed by separate teams using different technology stacks. This may result in increased efforts to maintain consistency and compatibility between components.
4. **Duplicate code and dependencies:** With each micro frontend potentially using its own set of libraries and frameworks, there might be duplicate code or dependencies across components. This can increase the overall application size and maintenance burden, making it crucial to manage shared resources effectively.
5. **Testing complexity:** As micro frontends involve numerous independent components, testing the application as a whole can be more complicated. Comprehensive testing strategies, including end-to-end testing and integration testing, must be established to ensure the overall system functions correctly.
6. **Team collaboration challenges:** Micro frontend teams often work autonomously, which can lead to silos and poor communication between teams. To avoid this, organizations need to establish clear communication channels and processes that foster collaboration and knowledge sharing across teams (Geers, 2020).
7. **Learning curve:** Implementing a micro frontend architecture may require team members to learn new technologies or approaches, which can initially slow down development. It is important to provide adequate training and support to ensure a smooth transition. (Geers, 2020).

2.5 When Micro Frontends are Beneficial?

Micro frontends make sense in a variety of situations, particularly when developing large-scale, complex web applications. Some key scenarios in which micro frontends are beneficial include:

1. **Large teams or multiple teams working on the same application**: When multiple teams or developers work on different parts of the application simultaneously, micro frontends allow them to develop and deploy their components independently without affecting the work of others.
2. **Diverse technology stacks**: If the application uses different technologies, frameworks, or libraries across its various components, micro frontends enable individual teams to choose the most suitable tools for their specific tasks without imposing constraints on other teams.
3. **Scalability and maintainability**: Micro frontends can help improve scalability and maintainability of an application by breaking it down into smaller, manageable units. This allows for easier updates, more efficient bug fixes, and streamlined performance optimization.
4. **Incremental migration or modernization**: If there's a need to gradually migrate or modernize an existing monolithic frontend application, adopting a micro frontend approach can make the process more manageable. Parts of the application can be replaced incrementally with new micro frontends without affecting the entire system.
5. **Faster deployment and continuous delivery**: Since micro frontends can be developed, tested, and deployed independently, they facilitate faster deployment cycles and support continuous delivery practices. This enables teams to release new features or updates more frequently and with less risk.

However, micro frontends might not be the best fit for every project. For small applications or projects with a single, small team, the added complexity of managing micro frontends could outweigh their benefits. It is essential to evaluate the specific needs of the project before deciding whether to adopt a micro frontend architecture. (Mezzalira, 2021)

3 TECHNICAL IMPLEMENTATIONS OF MICRO FRONTENDS

3.1 Composition via Ajax

In this approach, the main application (also known as the "app shell" or "container") loads individual micro frontends asynchronously using Ajax requests (Geers, 2020).

3.1.1 Overview

The main application (container) is responsible for managing the overall layout and coordinating the loading of micro frontends. When the container app loads, it sends Ajax requests to fetch the necessary micro frontends' content, which could be HTML, CSS, and JavaScript.

The micro frontends are developed and deployed independently, each with its own endpoint or URL from where the container app can fetch the content. As the Ajax requests complete, the container app injects the fetched content into the appropriate placeholders or sections within the main layout. The micro frontends are initialized and rendered within the container, and they begin to handle user interactions and other responsibilities (Mezzalira, 2021).

3.1.2 Benefits

Ajax offers several relevant and valuable benefits:

1. **Decoupling**: Each micro frontend can be developed, deployed, and updated independently, reducing the risk of affecting other parts of the application.
2. **Flexibility**: Teams can choose their preferred frameworks, libraries, and tools for developing each micro frontend without constraints.
3. **Parallel development:** Multiple teams can work simultaneously on different micro frontends, accelerating the development process.

3.1.3 Drawbacks

However, implementing ajax does come with some inherent challenges that have been identified over the years:

1. **Performance:** Multiple Ajax requests can increase the initial loading time of the application, especially if there are many micro frontends or if they are not optimized. Techniques such as caching, lazy loading, or server-side rendering can help mitigate these issues.
2. **Increased complexity:** Managing the composition and communication between micro frontends can be more complex than working with a monolithic frontend, especially when handling shared states or coordinating events across components.
3. **Potential for inconsistencies:** If not managed carefully, different micro frontends may have inconsistent user experiences, styles, or behavior, leading to a fragmented UI.

3.2 Server-Side Composition

Server-side composition is another approach to implement micro frontends. In this method, the assembly of the different micro frontends into a single, cohesive user interface happens on the server before the content is sent to the client's browser.

3.2.1 Overview

The main application (container) serves as the entry point and manages the overall layout, as well as the routing and coordination of micro frontends. When a user requests a specific page or view, the container app sends requests to each micro frontend's endpoint on the server-side.

Each micro frontend is developed and deployed independently, with its own server-side endpoint, which generates and returns the required HTML, CSS, and JavaScript. The container app receives the content from the micro frontends, composes the final page by combining the content in the appropriate layout, and sends

the resulting HTML, CSS, and JavaScript to the client's browser. The browser renders the composed page, and the micro frontends handle user interactions and other responsibilities as needed (Newman, 2015).

3.2.2 Benefits

The server-side composition approach for managing micro frontends delivers numerous key advantages:

1. **Improved initial load performance:** As the content is composed and delivered in a single server-side response, there are fewer network requests and less client-side rendering overhead, leading to faster initial load times.
2. **Simplified browser compatibility:** Since the composition occurs on the server, there is less reliance on advanced browser features, which can be helpful for ensuring compatibility with older browsers.
3. **Better SEO:** Search engines can more easily crawl and index server-rendered content, which can lead to improved search engine optimization (SEO).

3.2.3 Drawbacks

However, it's also important to be aware of the inherent challenges associated with this method, which include:

1. **Server-side complexity:** Managing the composition and communication between micro frontends on the server can be complex, especially when handling shared state or coordinating events across components.
2. **Less dynamic behavior:** As the composition takes place on the server, updating or modifying the content may require additional server requests, which can lead to a less dynamic user experience compared to client-side composition approaches.
3. **Scalability:** Increased server-side processing can put additional load on the server infrastructure, which may impact scalability.

3.3 Composition via Webpacks's Module Federation

Webpack's Module Federation is a modern approach to implementing micro frontends, introduced in Webpack 5. It enables loading and sharing of code across different JavaScript applications at runtime, making it a powerful tool for composing micro frontends (Manfred, 2020).

3.3.1 Overview

The main application (container) is responsible for managing the overall layout, routing, and coordination of micro frontends. Each micro frontend is developed and deployed independently as a separate application, exposing specific parts of its code (components, functions, etc.) as "federated modules."

The container application is configured with Webpack to consume these federated modules from the micro frontends. This includes specifying the remote entry points, module names, and other necessary configuration details. When the container app loads, it fetches and combines the micro frontends' federated modules dynamically at runtime, integrating them into the main application. The micro frontends are initialized, rendered within the container, and begin handling user interactions and other responsibilities (Manfred, 2020).

3.3.2 Benefits

Embracing the Module Federation approach in managing micro frontends provides a variety of distinct benefits that can significantly enhance the development process and the quality of the end product:

1. **Decoupling:** Each micro frontend can be developed, deployed, and updated independently, allowing for better separation of concerns and reduced risk of affecting other parts of the application.
2. **Code sharing:** Module Federation enables efficient sharing of common dependencies and libraries across micro frontends, reducing the overall application size and improving performance.

3. **Flexibility:** Teams can choose their preferred frameworks, libraries, and tools for developing each micro frontend without constraints.

3.3.3 Drawbacks

However, it is crucial to take into account the potential challenges associated with Module Federation, as outlined below:

1. **Complexity:** Managing the composition and communication between micro frontends using Module Federation can be complex, especially when handling shared state or coordinating events across components.
2. **Webpack dependency:** This approach relies on Webpack 5 or newer, which may not be compatible with older applications or projects using different bundlers.
3. **Performance:** If not optimized properly, loading multiple micro frontends and their dependencies can impact the initial loading time of the application. Techniques such as code splitting, lazy loading, and caching can help mitigate these issues.

3.4 Communication Patterns

Communication patterns are essential for ensuring that the different components of a micro frontend architecture can efficiently share data and events. Some common communication patterns in micro frontends include:

1. **Custom Events:** Custom events allow micro frontends to communicate with each other through the browser's native event system. Components can emit custom events when they need to share information or trigger actions in other components, and listeners can be set up to react to these events (Geers, 2020).
2. **Shared State:** By using a shared state (for example, Redux or MobX), micro frontends can manage their state in a centralized store. This allows com-

ponents to read and update the state, ensuring consistency across the application. However, this approach can lead to tight coupling if not implemented carefully.

3. **API-based communication:** Micro frontends can communicate by consuming and providing APIs. Each component exposes an API to interact with its functionality, allowing other components to use this interface to request data or perform actions. This approach promotes loose coupling and enables easier integration of third-party components.
4. **Message Bus / Publish-Subscribe:** A message bus or publish-subscribe pattern allows micro frontends to communicate asynchronously by publishing messages to channels or topics. Components can subscribe to these channels to receive updates or events. This pattern enables decoupling, as components do not need to know about each other directly (Geers, 2020).
5. **Local Storage / Session Storage:** Components can use browser storage mechanisms (local or session storage) to share data between them. This approach can be useful for persisting state between page refreshes or different browser tabs.
6. **Web Components:** Web Components are a set of browser APIs that allow the creation of custom, reusable HTML elements. By encapsulating their functionality, micro frontends can use web components to communicate with each other, allowing for easy integration and a consistent user experience.
7. **GraphQL:** GraphQL is a query language and runtime that enables components to request only the data they need from a backend service. This can help streamline communication between micro frontends, as well as between the frontends and backend services.
8. **Observables:** Observables, like those provided by the RxJS library, allow micro frontends to communicate by subscribing to data streams. This approach supports reactive programming and enables components to react to changes in data sources or user input.

Each communication pattern has its benefits and trade-offs, and the choice of pattern depends on the specific requirements of the application. Often, a combination of these patterns can be used to ensure effective communication between micro frontends.

4 MIGRATING TO MICRO FRONTENDS

4.1 Concepts

Undertaking the migration of a substantial project from one architectural framework to another can be a daunting and often expensive endeavor. Various approaches exist, each with their advantages and disadvantages. To effectively manage expectations and budget, it is crucial to have a clear understanding of the complexity and effort involved in such a migration. However, accurately estimating the required resources can be challenging when a team lacks experience with the target architecture. Experimenting with the new technology in a sandbox environment can help alleviate uncertainties, and the examples in this book can serve as an excellent foundation for these explorations (Mezzalira, 2021).

Micro frontends and their user interface integration techniques are particularly useful for facilitating gradual migrations. The micro frontend approach is well-suited for constructing and incorporating a proof of concept, which can even be tested within a live production application (Flower, 2019).

Here are the key steps involved in migrating to micro frontends:

1. **Analyze the current architecture**: Begin by understanding the existing monolithic frontend application. Identify its strengths, weaknesses, and dependencies.
2. **Define domain boundaries:** Break down the application into smaller, logical, and self-contained components based on business domains or functionality. These smaller components will become the micro frontends.
3. **Choose a composition strategy:** Decide how micro frontends will be composed and integrated. There are several approaches, such as client-side composition (using tools like Webpack Module Federation), server-side composition (using tools like Edge Side Includes), or build-time composition.
4. **Establish communication channels**: Determine how micro frontends will communicate with each other and with backend services. This may involve

setting up APIs, message buses, or using shared state management libraries.

5. **Implement a development framework:** Choose a development framework that supports micro frontends or create a custom one. This may involve adopting component libraries, build tools, and deployment pipelines that facilitate independent development and deployment of micro frontends.
6. **Migrate incrementally:** Rather than performing a complete overhaul, migrate parts of the application to micro frontends one by one. This approach minimizes risk and allows for iterative improvements.
7. **Establish team structures:** Organize development teams around micro frontends, empowering them to make decisions related to their specific domain. This fosters autonomy and enables faster development cycles.
8. **Ensure consistency:** Create guidelines for UI design, coding practices, and performance standards to maintain a consistent look, feel, and behavior across micro frontends.
9. **Monitor and optimize:** Continuously monitor the performance and user experience of micro frontends, identifying areas for improvement and optimization.
10. **Maintain and update:** Regularly update and maintain the micro frontends to ensure they remain secure, performant, and aligned with evolving business requirements.

4.2 Common Strategies

4.2.1 Strangler Pattern

In this approach, new features or components are developed as micro frontends while the existing monolithic application is gradually decommissioned. The new micro frontends are integrated into the monolith, progressively replacing parts of the original application (Mezzalira, 2021).

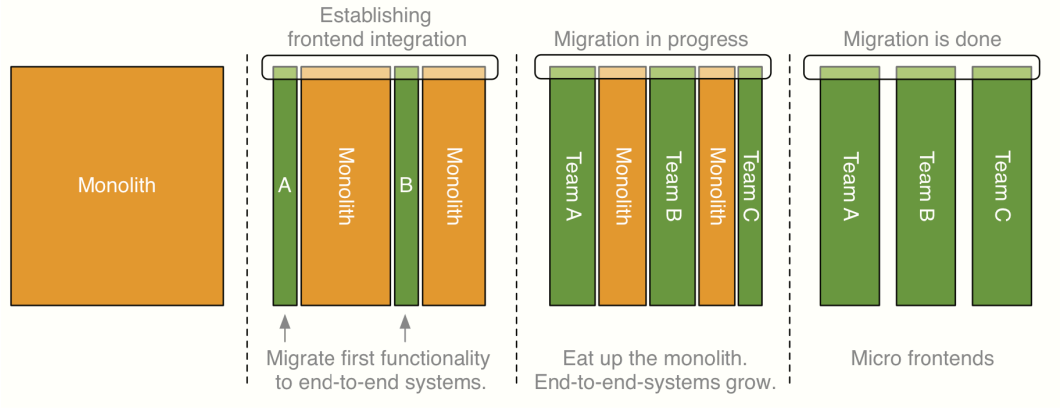

Figure 2. Diagram showing Strangler Pattern

Benefits of the strangler pattern include

- Minimizes risk, as changes are incremental and do not disrupt the entire application.
- Allows teams to work independently on specific micro frontends, improving development efficiency.
- Simplifies testing and deployment, as only the affected micro frontends need to be updated.

Risks of the strangler pattern comprise the following

- Demands careful planning and coordination to avoid tight coupling between micro frontends and the monolith.
- Requires a longer transition period, as the monolith and micro frontends coexist during the migration process.
- May introduce temporary complexity, as developers need to maintain and understand both the monolith and the new micro frontends.

4.2.2 Parallel Run

This strategy involves developing the entire micro frontend architecture parallel to the existing monolithic application. Once the new architecture is complete and tested, the monolithic application is replaced in its entirety (Mezzalira, 2021).

Benefits of the parallel run pattern include

- Promotes consistency in UI/UX design and code quality across micro frontends.
- Allows developers to focus on the new architecture without managing the complexities of integrating with the monolith.
- Facilitates a faster transition once the new architecture is ready for deployment.

Risks of the parallel run pattern comprise the following

- Requires significant upfront investment in development and testing before realizing any benefits.
- Introduces the risk of a "big bang" deployment, which may lead to unexpected issues and downtime.
- Can be resource-intensive, as teams must develop and maintain two separate codebases during the migration process.

4.2.3 Iterative Replacement

In this approach, the monolithic application is incrementally replaced by micro frontends, one component or feature at a time. As each micro frontend is completed, it replaces the corresponding part of the monolith (Mezzalira, 2021).

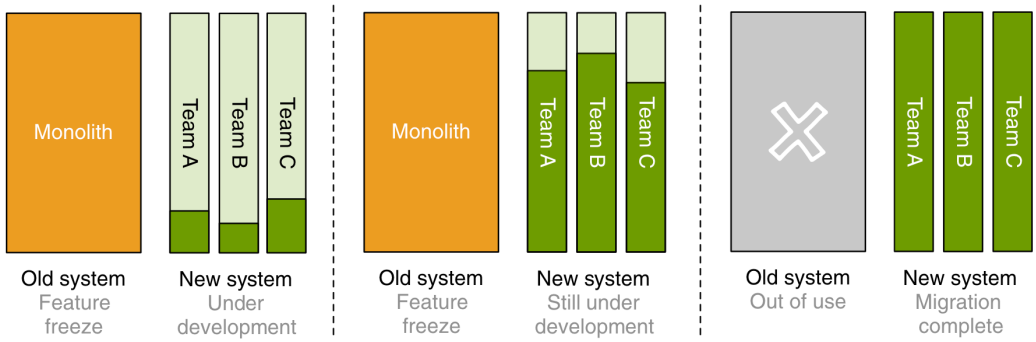

Figure 3. Diagram showing Iterative Replacement Pattern

Benefits of the iterative replacement pattern include

- Reduces risk, as changes are incremental and can be tested and deployed individually.
- Allows for continuous improvement and adaptation throughout the migration process.
- Minimizes the impact on end-users, as the application remains functional during the migration.

Risks of the iterative replacement pattern comprise the following

- Requires careful coordination between teams to ensure smooth integration and avoid tight coupling.
- May lead to temporary inconsistencies in UI/UX design, as both the monolith and micro frontends coexist during the migration.
- Can be time-consuming, as the migration process involves multiple stages and iterations.
- When selecting a migration strategy for transitioning from a monolithic application to micro frontends, it is essential to consider the unique requirements and constraints of the project, as well as the resources and expertise available within the team.

5 WEBPACK MODULE FEDERATION

Module Federation, a key feature in Webpack 5, plays a crucial role in our practical case. It makes it simple to transform compile-time dependencies into runtime dependencies. This means that the application treats both remote modules and its own internal components in the same way.

```
{
  path: 'tasks',
  pathMatch: 'full',
  loadChildren: () => {
    return import('./tasks/tasks.module').then(m => m.TasksModule);
  },
  canActivate: [ModulesGuard]
},
{
  path: 'tasks',
  pathMatch: 'full',
  loadChildren: () => loadRemoteModule( options: {
    type: 'manifest',
    remoteName: 'http://localhost:4201/remoteEntry.js',
    exposedModule: './TasksModule'
  }).then(m => m.TasksModule),
  canActivate: [ModulesGuard]
}
```

Figure 4. Compile-time and Runtime Dependencies

By using Module Federation, developers can improve their project's modularity. They can manage how code is shared between different applications. This is done using "host" and "remote" applications, where the host app dynamically imports and uses the functionality provided by the remote app. As a result, there is no need to include the same code or libraries more than once. This reduces the total bundle size and improves performance (Manfred, 2020).

Incorporating Module Federation is a big step forward in software engineering. It helps develop complex systems that are connected, while still following principles of maintainability and reusability. By easily turning compile-time dependencies into runtime dependencies, Module Federation ensures that the application treats remote modules and its own components in the same way. This makes collaboration between developers smoother and encourages the creation of scalable, efficient software solutions (Manfred, 2020).

Module Federation is designed with flexibility and adaptability in mind, catering to a wide range of use cases and environments. The following are some key points that illustrate the capabilities and features of Module Federation:

1. **Module Type Support**: Module Federation allows developers to expose and consume any module type supported by Webpack. This enables seamless integration with various technologies and libraries, promoting a versatile development experience.
2. **Optimized Chunk Loading**: In web environments, chunk loading is designed to fetch all necessary resources in parallel, ensuring a single round-trip to the server. This approach significantly reduces loading times and enhances the overall performance of applications.
3. **Control from Consumer to Container**: Module Federation provides a one-directional control mechanism, where consumers can override container modules. However, sibling containers cannot override each other's modules, preserving the autonomy and isolation of individual modules.
4. **Environment-Independent Concept**: The Module Federation concept is not limited to a specific environment; it is usable in web, Node.js, and other environments, making it a versatile solution for diverse software projects.
5. **Relative and Absolute Requests in Shared Modules**: Shared modules with relative and absolute requests will always be provided, even if not used. These requests will be resolved relative to the *config.context* and do not require a *requiredVersion* by default.

6. **Module Requests in Shared Modules**: Shared module requests are provided only when they are used. They will match all used equal module requests in the build, provide all matching modules, and extract the *requiredVersion* from the *package.json* at the specified position in the graph.
7. **Nested Node Modules**: Module Federation can handle multiple different versions of nested *node_modules*, allowing developers to provide and consume various versions when required.
8. **Module Requests with Trailing Slash**: Module requests with a trailing slash in shared modules will match all module requests with the corresponding prefix. This feature enhances the flexibility and compatibility of the Module Federation system, ensuring a more streamlined development experience.

In summary, Module Federation, a powerful feature in Webpack 5, enables the dynamic and seamless sharing of code between distinct applications or micro-frontends. It simplifies the transformation of compile-time dependencies into runtime dependencies, treating remote modules and internal components in the same manner. This approach promotes a scalable and maintainable architecture, fostering collaboration among developers and enhancing overall software performance.

6 DEMO APPLICATION

In order to prevent the disclosure of the company's private information, I built a demo application. I will explain the process of migrating from a monolithic application to micro frontends in this chapter. The demo application is neither based on nor resembles the actual system, but it does utilize the same or similar underlying open-source technologies.

One of the main challenges the real-world apps encounter is ensuring seamless and secure communication between the micro frontends. This demo simplifies this part to a primitive message bus, utilizing Broadcast Channel API, a relatively recent standard, offers another method for communication. Functioning as a publish/subscribe system, it facilitates communication across tabs, windows, and even iframes within the same domain.

```
sayHello() : void {
  const chan : BroadcastChannel  = new BroadcastChannel( name: 'test_channel');
  const eventButton : Element | null  = document.querySelector( selectors: '#eventButton');
  eventButton!.addEventListener( type: 'click',  listener: () : void  => {
    chan.postMessage('hello');
  });
}
```

Figure 5. Example usage of BroadcastChannel API

In our context, each micro frontend could initiate a connection to a central channel, like 'test_channel', and consequently receive notifications from other micro frontends. This allowed for more manageable and efficient coordination among the different micro frontends.

Although the demo application does not exactly mirror the actual Fliq system, the underlying principles and technologies remain the same. Hence, this step-by-step walkthrough provides a general blueprint that can be tailored to meet the specific needs of other similar migrations.

6.1 Notes on the Initial Application

The application in question is a simple modular application designed to monitor and control various aspects of industrial automation in a factory or industrial process. It mocks real-time process monitoring, control panel functionality, notifications and alerts, historical data analysis, and predictive maintenance capabilities. However, the mocked data can easily be substituted with actual data provided by an API.

The application is build using Angular 16 and consist of a shell application and 5 modules:

1. The **Real-Time Process Monitoring** module displays live data feeds from different areas of the industrial process, such as production line status, temperature readings, pressure levels, and machine operation status. It provides graphical representations such as charts, gauges, and diagrams to visualize the current operational status effectively.
2. The **Control Panel** module enables operators to interact with the industrial process by starting or stopping machines, adjusting settings, and triggering specific actions. It also supports more complex procedures such as scheduling maintenance or changing production parameters.
3. The **Notifications and Alerts** module generates and displays alerts and notifications based on certain conditions or thresholds in the industrial process. The alerts are color-coded or categorized based on severity or type, allowing operators to quickly identify critical issues and take appropriate actions.
4. The **Historical Data Analysis** module allows operators to view and analyze past operational data. It provides features for generating reports, making comparisons over time, identifying trends, and gaining insights into the performance of the industrial process.

5. The **Predictive Maintenance** module utilizes machine learning algorithms to predict when machines or components might fail based on historical and real-time data. It presents these predictions graphically and suggests actions to prevent downtime, enabling proactive maintenance and minimizing disruptions in the industrial process.

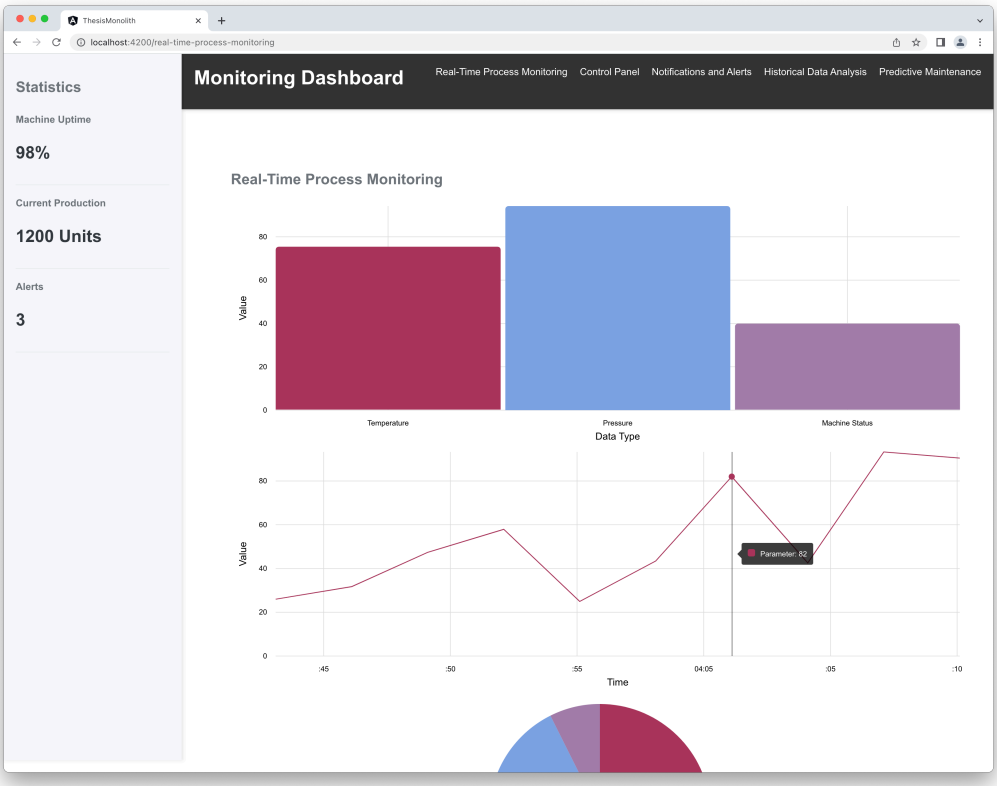

Figure 6. The main page of the monolithic Demo application

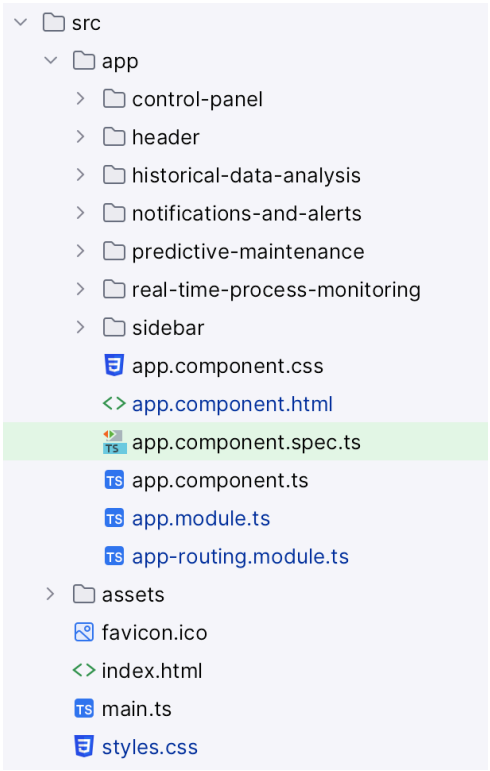

Figure 7. The file structure of the monolithic Demo application

In its initial state, the application is monolithic. Now, we review the necessary steps to convert this application to micro frontends using Webpack Module Federation.

6.2 Transition

The transition to micro frontends with splitting the monolith into the shell application and 5 standalone modules. We will use a monorepo approach, where the code for the shell application and the individual modules will be organized and managed within a single repository.

A monorepo, short for monolithic repository, is a software development approach that involves storing multiple related projects or modules within a single repository. In the context of our application, the monorepo will contain the code for both the shell application and the standalone modules.

Using a monorepo offers several advantages. It allows for better code sharing and reuse between different parts of the application. Developers can easily navigate and work on different modules within a unified codebase. It simplifies versioning and dependency management since all the modules share the same version control and dependency configurations.

In our case, the monorepo will provide a convenient structure for managing and coordinating the development of the shell application and its modules. It will facilitate the sharing of common code, utilities, and configurations among the modules while maintaining their independence.

First, we create a shell app and configure Webpack:

```
new ModuleFederationPlugin({
    library: { type: "module" },

    remotes: {
        "real-time-process-monitoring": "http://localhost:4201/remoteEntry.js",
        "control-panel": "http://localhost:4202/remoteEntry.js",
        "notifications-and-alerts": "http://localhost:4203/remoteEntry.js",
        "historical-data-analysis": "http://localhost:4204/remoteEntry.js",
        "predictive-maintenance": "http://localhost:4205/remoteEntry.js",
    },

    shared: share( shareObjects: {
      "@angular/core": { singleton: true, strictVersion: true, requiredVersion: 'auto' },
      "@angular/common": { singleton: true, strictVersion: true, requiredVersion: 'auto' },
      "@angular/common/http": { singleton: true, strictVersion: true, requiredVersion: 'auto' },
      "@angular/router": { singleton: true, strictVersion: true, requiredVersion: 'auto' },

      ...sharedMappings.getDescriptors()
    })
}),
```

Figure 8. Shell application's webpack.config.ts

Our remotes are not dynamic in nature, which means we can declare them straight away using their identifiers and URLs, as these identifiers and URLs remain constant.

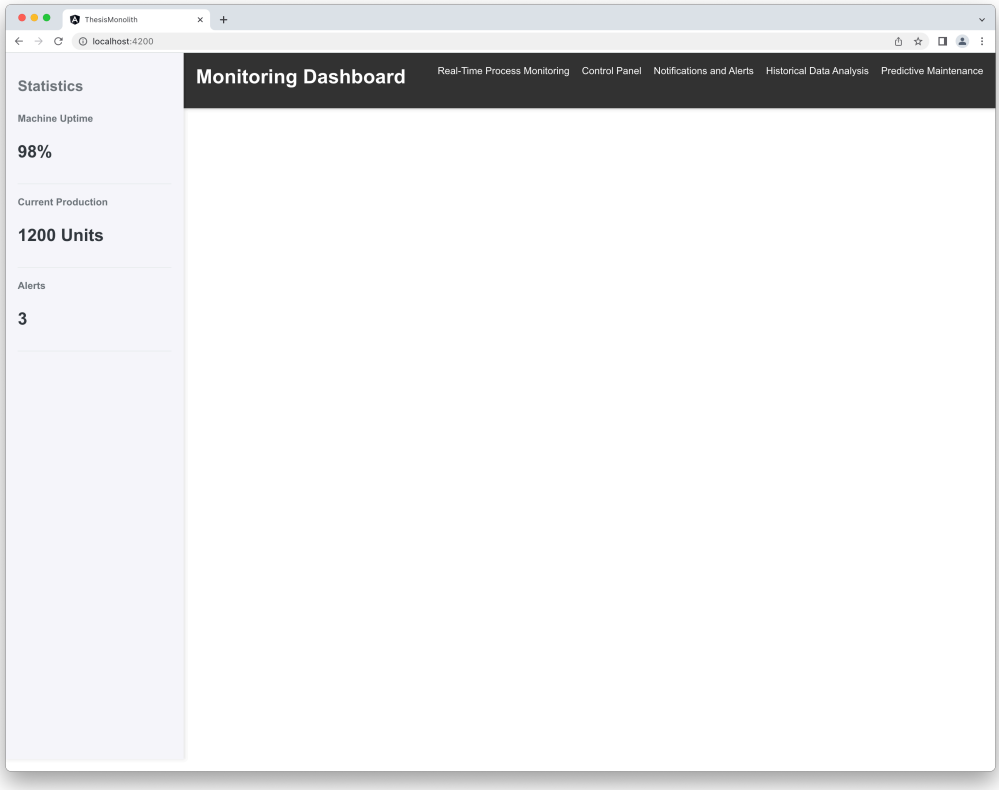

Figure 9. Initial state of the Shell application

The shell application is currently empty, as we have solely declared the micro frontends, but they are yet to be built and run within the application. This is the initial stage of our transition towards a micro frontend architecture.

To bring the micro frontends to life, we need to follow the necessary steps to build and run each module individually. This involves configuring the development environment, setting up the appropriate build processes, and establishing communication channels between the shell application and the micro frontends.

As we are running a monorepo, creating a new application is as easy as running the following command:

```
ng generate application app-name
```

This command will automatically generate a new application within the monorepo, and adjust the necessary configuration files accordingly. The Angular CLI takes care of scaffolding the required files and directories specific to the new app.

By following this approach, all applications within the monorepo share a common set of configurations and dependencies, making it easier to manage and maintain multiple applications in a single codebase. Additionally, the monorepo structure promotes code sharing and reusability among the different applications and libraries within the project.

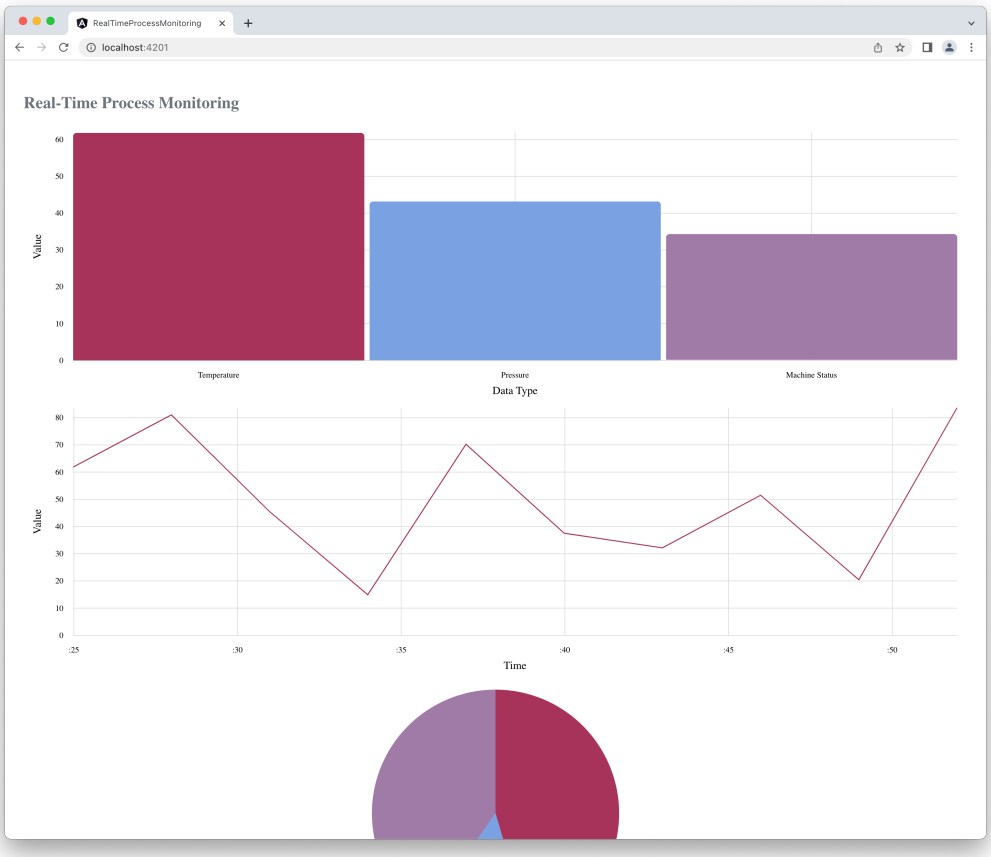

Figure 10. Real-time monitoring micro frontend

The module can be used either as a standalone application or seamlessly integrated into the shell application. Even more impressive is that it has the flexibility to be embedded into multiple shells simultaneously. This means that the module can be utilized in various scenarios, whether it is operating independently, integrated into a single shell, or incorporated into multiple shells at the same time.

In addition to integrating the module into the shell application, we also need to adjust the Webpack configuration of the micro frontend. This involves declaring which modules are exposed and can be consumed by other parts of the application.

By configuring the Webpack module federation settings, we define the specific modules and their corresponding entry points that will be made available for consumption. This allows the shell application or other micro frontends to dynamically import and utilize the exposed modules.

Properly adjusting the Webpack configuration ensures that the necessary dependencies and functionality from the micro frontend are accessible to the rest of the application. It establishes a clear boundary between the exposed modules and encapsulates their internal implementation details, promoting modularity and encapsulation.

```
new ModuleFederationPlugin({
    library: { type: "module" },

    name: "realTimeProcessMonitoring",
    filename: "remoteEntry.js",
    exposes: {
        './Module': './projects/real-time-process-monitoring/src/app/monitoring/monitoring.module.ts',
    },

    shared: share( shareObjects: {
      "@angular/core": { singleton: true, strictVersion: true, requiredVersion: 'auto' },
      "@angular/common": { singleton: true, strictVersion: true, requiredVersion: 'auto' },
      "@angular/common/http": { singleton: true, strictVersion: true, requiredVersion: 'auto' },
      "@angular/router": { singleton: true, strictVersion: true, requiredVersion: 'auto' },

      ...sharedMappings.getDescriptors()
    })

}),
```

Figure 11. Micro frontend webpack.config.ts

We need to update the shell router to effectively turn the compile-time dependencies into dynamic runtime-dependencies. To accomplish this, we will make use of Angular's router's loadChildren function. This function allows us to dynamically

load modules at runtime, which fits perfectly with our micro frontend architecture.

Traditionally, the loadChildren function was used for lazy-loading Angular modules to improve the initial load time of an application. However, we can repurpose this function to load our micro frontend modules.

6.3 Micro Frontends as Web Components

One advantage of the microfrontend architecture is the ability to use external components, or components made by a third party. However, it is important to understand that modern web frameworks are not designed to operate with multiple versions of themselves at runtime. This raises the question of isolating these components. This isolation can be achieved using Web Components.

Web Components, a set of web platform APIs, allow the creation of custom, reusable, encapsulated HTML tags for use in web pages and web apps. They are an integral part of the browser and so they do not need additional libraries or frameworks to run and can maintain compatibility with all modern frameworks.

In the context of microfrontends, Web Components can encapsulate the implementation of each micro-app, ensuring they operate independently, without interference. For example, one can load two components using different versions of the same framework and they will coexist without conflicts, because their implementation details are scoped within the custom elements.

Let us consider how we can integrate an external micro frontend application built using different version of Angular into the shell application. The process would require 3 steps:

1. **Wrap the micro frontend in a Web Component**

Wrapping can be done using Angular Elements.

```
export class AppModule {
  constructor(private injector: Injector) {}

  ngDoBootstrap() {
    const el = createCustomElement(AppComponent, { injector: this.injector });
    customElements.define('angular-element', el);
  }
}
```

Figure 12. Exposing standalone Angular application as a Web Component

This code snippet overrides the default Angular bootstrap process by providing a custom bootstrapper. Then it registers a new Web Component with the 'angular-element' tag directly using the browser's API. Consequently, the browser will employ this element whenever the <angular-element></angular-element> tag is found in the application.

2. Expose the module using Module Federation

To use the micro frontend in the shell application, we need to expose its Web Component using Module Federation. An example configuration might look like this:

```
plugins: [
  new ModuleFederationPlugin({
      library: { type: "module" },

      name: "externalMfe",
      filename: "remoteEntry.js",

      exposes: {
        './web-components': './src/bootstrap.ts',
      },

      shared: share({
        "@angular/core": { requiredVersion: 'auto' },
        "@angular/common": { requiredVersion: 'auto' },
        "@angular/common/http": { requiredVersion: 'auto' },
        "@angular/router": { requiredVersion: 'auto' },

        ...sharedMappings.getDescriptors()
      })

  }),
```

Figure 13. Exposing a Web Component using Webpack

Unlike previous Webpack configurations, we do not enforce a strict versioning of the shared modules because, in this example, the goal is to share two different versions of Angular. The 'auto' parameter obtains the package versions from *package.json and* searches for them at runtime. If no match is found, it defaults to its own version.

3. Load Custom Elements into the shell application.

We can use the same lazy loading mechanism that we use for loading standard Angular components. In the example, the WebComponentWrapper loads the Web Component using Module Federation with the provided parameters. This wrapper also creates a custom HTML element.

```
{
  path: 'external-angular',
  component: WebComponentWrapper,
  data: {
    type: 'module',
    remoteEntry: 'http://localhost:5200/remoteEntry.js',
    exposedModule: './web-components',
    elementName: 'angular-element'
  } as WebComponentWrapperOptions
}
```

Figure 14. Wrapping an external Web Component

Wrapper essentially acts like a module here, rendering the external Web Component inside itself.

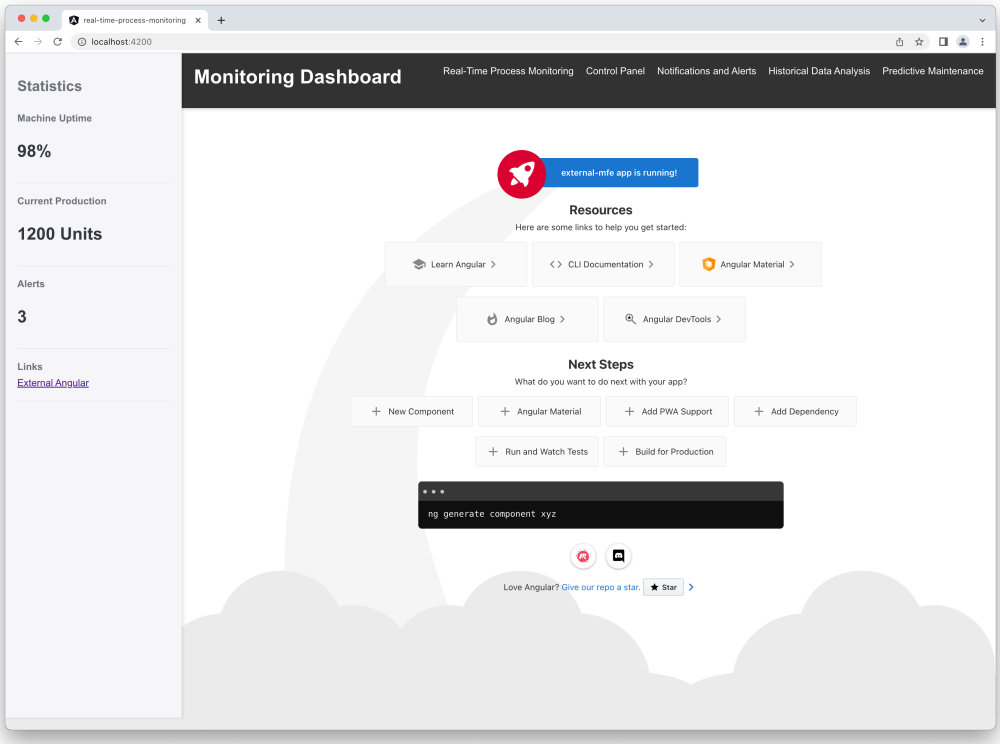

Figure 15. Angular 15 running inside Angular 16

6.1 Using Other Frameworks

It is also possible to load micro frontends based not just on different version of the same framework, but built using completely different set of technologies, be it another framework or vanilla JavaScript. For example, the process of integrating a React-based component into the demo application is essentially the same as integrating an Angular-based Web Component.

The first step in this integration is to expose the React component as a Web Component, similar to how we exposed the Angular component previously. The Web Component then needs to be registered in the host application and made available for use. Once the React component is registered, it can be loaded dynamically at runtime just like any other component.

```
new ModuleFederationPlugin({
    name: "react",
    library: { type: "var", name: "react" },
    filename: "remoteEntry.js",
    exposes: {
        './web-components': './app.js',
    },
    shared: ["react", "react-dom"]
}),
```

Figure 16. Exposing React component as a Web Component

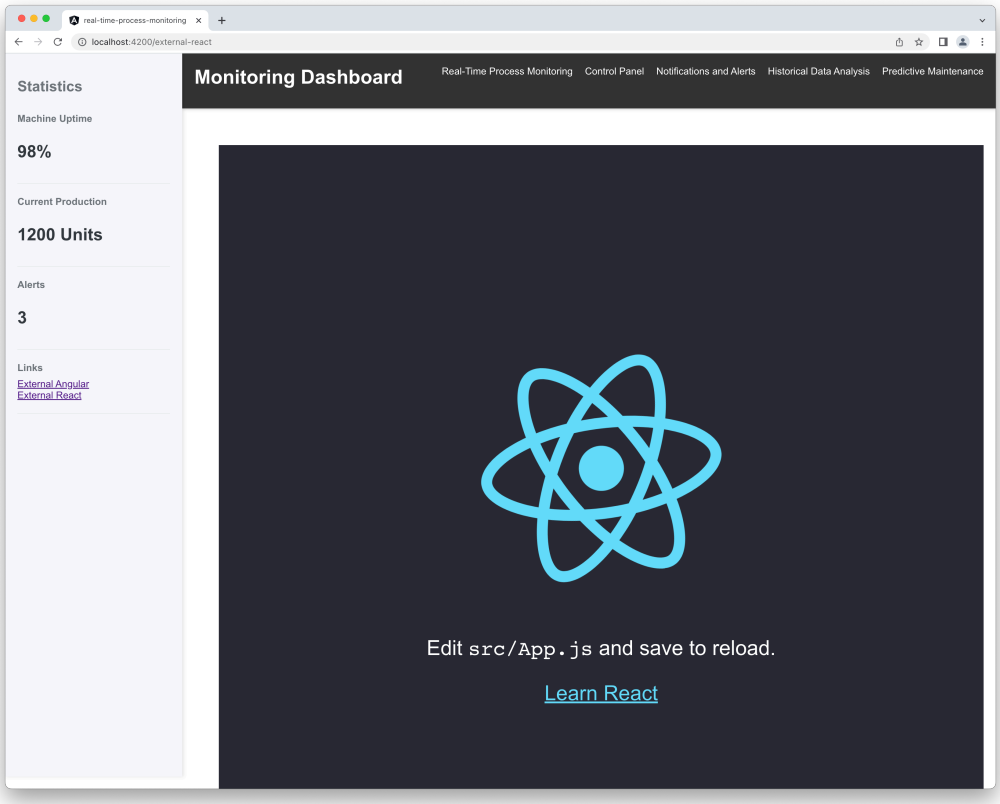

Figure 17. React running inside Angular.

7 CONCLUSION

In conclusion, this thesis has comprehensively explored the concept of micro frontends and how they can be utilized to effectively scale a software platform. Throughout the investigation, it has become apparent that the micro frontend architecture offers significant advantages in terms of development speed, team autonomy, technology freedom, and overall scalability.

The research has revealed that the use of micro frontends allows for more manageable codebases, as each frontend can be developed, tested, and deployed independently of others. This structure significantly reduces the risk of conflicts, allows for more efficient code management, and enables teams to work autonomously without the need for extensive coordination.

Moreover, the ability to employ different technologies across various frontends without any interdependencies significantly enhances technological freedom. This allows individual teams to select the technologies that best suit their specific needs, enhancing innovation, and expediting delivery.

Most importantly, micro frontends are a key enabler for scalability. The decoupled nature of micro frontends means that as the software platform grows, additional frontends can be added with minimal impact on the existing system. This provides a high degree of scalability, a feature that is particularly important in today's rapidly evolving digital landscape.

Despite the many benefits of micro frontends, it is important to note that the architecture is not a one-size-fits-all solution. Careful consideration needs to be given to factors such as team size, project complexity, and specific business requirements. However, for larger, complex projects, where scalability, team autonomy, and technological freedom are paramount, micro frontends provide a compelling option.

REFERENCES

Flower, Martin (2019) Micro Frontends. Accessed 23.05.2023. Available at https://martinfowler.com/articles/micro-frontends.html

Geers, Michael (2020) Micro frontends in Action. Available at https://learning.oreilly.com/library/view/micro-frontends-in/9781617296871/

Gilbert, John (2021) Software Architecture Patterns for Serverless Systems. Available at https://learning.oreilly.com/library/view/software-architecture-patterns/9781800207035/

Grijzen, Erik (2019) Micro Frontend Architecture Building an Extensible UI Platform". Accessed 23.05.2023. Available at https://www.youtube.com/watch?v=9Xo-rGUq-6E

Lemon, Steven (2020) Problems with Micro-frontends. Accessed 23.05.2023. Available at https://medium.com/swlh/problems-with-micro-frontends-8a8fc32a7d58

Mezzalira, Luka (2021) Building Micro-Frontends. Available at https://learning.oreilly.com/library/view/building-micro-frontends/9781492082989/

Newman, Sam (2015) Building Microservices. Available at https://www.oreilly.com/library/view/building-microservices/9781491950340/

Schottner, Lothar (2021) The Art of Micro Frontends. Available at https://learning.oreilly.com/library/view/the-art-of/9781800563568/

Steyer, Manfred (2020) The Microfrontend Revolution: Module Federation with Angular. Accessed 23.05.2023. Available at https://www.angulararchitects.io/en/aktuelles/the-microfrontend-revolution-part-2-module-federation-with-angular/